# Arquitetura de Computadores 2024/25

# TPC4

This homework consists of a programming exercise to be completed **in a group of no more than two students**. You can discuss general doubts with colleagues, but the solution and the writing of the code must be strictly carried out by the members of the group. All solutions will be automatically compared, and cases of plagiarism will be punished according to current regulations.

**Deadline for submission: 7/6 (Saturday) at 18:00.**

## Dynamic memory allocation and *mmap* OS call

During the execution of a program, obtaining space in RAM is typically based on the C library functions malloc and free. The memory space used for this is called the heap, which is a continuous space that can be altered with the sbrk system call. The following figure summarizes the memory map of a process in an operating system like Linux:

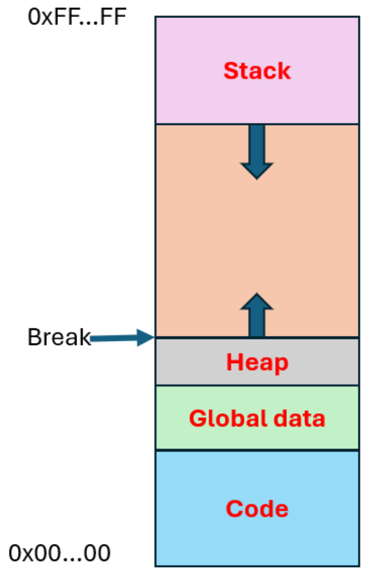

*Figure 1 - Memory map of a Linux process*

A process can obtain space in RAM through the *mmap* system call. This system call has a complex set of parameters that can be consulted using the command *man mmap*, but in the context of this work, we assume the set of parameters exemplified in the following C program:

```
#include <sys/mman.h>

size_t size = ... ; // size must be a multiple of the size of a page (4096)

void *addr = mmap( 0, size, PROT_READ | PROT_WRITE , MAP_PRIVATE  | MAP_ANONYMOUS, -1, 0 );
if( addr == (void *)-1) ){
        perror("mmap");
        return 1;
}
```

If the system call is successful, the memory map becomes:

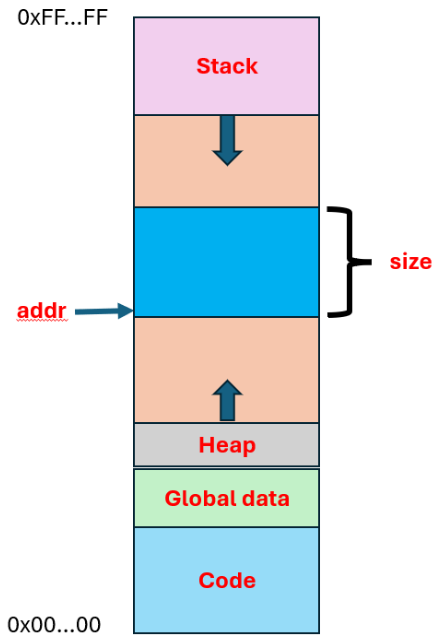

*Figure 2 - Memory map after using the mmap system call*

The process has access to the new area via C instructions such as:

```
char *pt = addr;

pt[5] = pt[22] + 5;  // reading and writing in the new allocated memory area
```

In this work, we will implement a set of operations (API) that allow the use of this memory area through operations similar to the malloc and free operations in the C library. The API is defined in the *myAlloc.h* file whose content is as follows:

```
typedef struct {
  void *base;                    // beginning of managed area
  void *top;                     // address of last byte allocate
  unsigned long int *limit;   // last address in the area allocated by mmap
} heap;

typedef struct{
  unsigned int available;
  unsigned int size;
} block;

void allocate_init (heap *h, unsigned long int maximumSize);
void *allocate (heap *h, unsigned int size);
void deallocate (void * p);
```

When a data block with *nBytes* is reserved with the *allocate()* operation, it has two parts:

- Metadata:
  - *available* field which is an *unsigned int* (4 bytes). It can have the value AVAILABLE (1) or UNAVAILABLE (0).
  - *size* field which is an *unsigned int* (4 bytes). Contains the size in bytes of the assigned zone.

- Data: *nBytes* that the caller of the allocate operation can use freely.

The following figure presents a block with 100 available bytes:

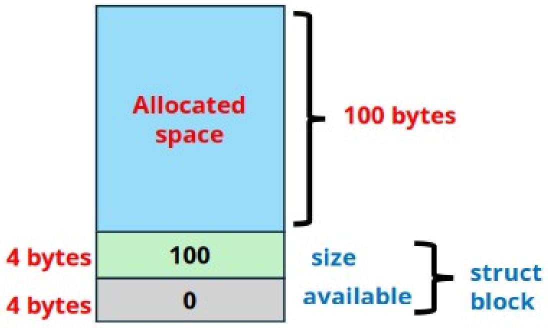

*Figure 3 - A memory block with its descriptor*

The state of the memory space is represented in the heap structure, which contains the following fields:

- *base:* initial address of the managed area
- *top:* address of the the first byte after the last byte allocated by a previous *allocate()* operation
- *limit:* address of the the first byte after the last address of the managed area

The following figure shows an example of a scenario after performing several *allocate()* and *deallocate()* operations:

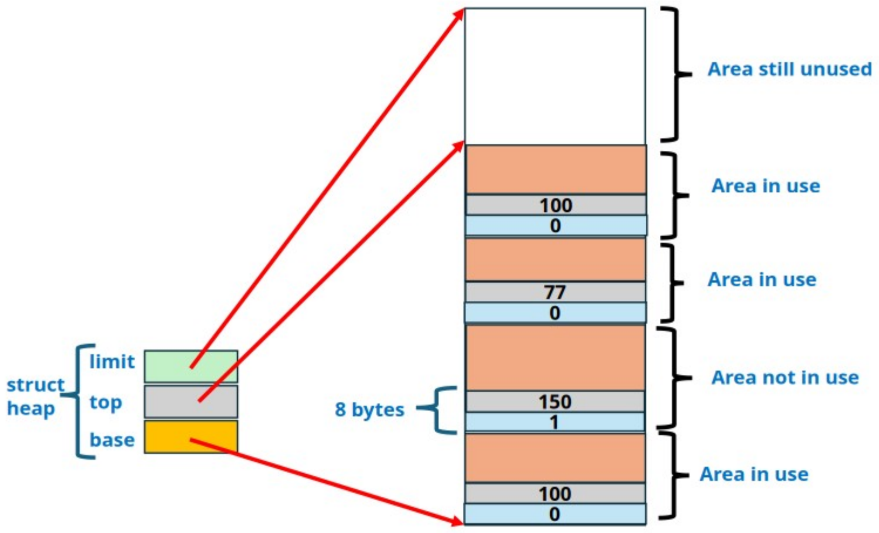

*Figure 4 - Overview*

## Work to be developed

In this work we want to implement, in x86_64 assembly, the *allocate()* and *deallocate()* functions whose behavior is described below.

### void *allocate (heap *h, unsigned int bytesToAllocate);

This function returns the address of a block with a size greater than or equal to *bytesToAllocate*. To achieve this, it traverses the memory described in Figure 4, starting at *h->base* and jumping to the next *block* structure.

The traversal ends when one of the following two conditions is met:

1. a *block* structure *b* is found which meets the conditions:

    *b->available == 1*
    *b->size >= bytesToAllocate*

    If more than one block matches these conditions, the block that has the lower value of field *size*, should be chosen.
    In this case, it returns to the user the address *b+sizeof(block)* and sets *b->available* to *0*.
2. if it exceeds the *h->top* value. In this case, a new block with size *bytesToAllocate* must be created. If successful, the procedure is the same as in 1); the *h* structure must be updated. If it is not possible to create the new *block* because it would exceed the address *h->limit*, the function should do nothing and return -1.

### void deallocate (void *address);

The received address is the data block (see Figure 3). The function will have to place the constant AVAILABLE (1) in the *available* field of the *block* structure.

## Available files

The following files are available on CLIP:

- the *myAlloc.h* file that defines the API to use.
- a complete C *main.c* program. This program is invoked with two arguments:

    *./main sizeOfMemory maxBlock*

    where *sizeOfMemory* is the size of the memory managed by the *allocate()* and *deallocate()* functions. *maxBlock* is the maximum size of the memory block used in the tests performed. The file includes the invocation of the *mmap()* system call which obtains the memory area to be managed; the *adjustToMultipleOfPageSize* function adjusts *sizeOfMemory* to the smallest multiple of the page size used by the x86-64. The part of the performed tests can be modified.
- a file with a skeleton of the program in assembly called my*Alloc*.s. You will need to complete this file by writing the code for the *allocate()* and *deallocate()* functions with the behavior described above.
- a *Makefile* that generates an executable file.

```
CC=gcc
CFLAGS=-g -z noexecstack
ASFLAGS=-g -gstabs
all: main
main: main.c myAlloc.o
	$(CC) $(CFLAGS) -static -o main main.c myAlloc.o
myAlloc.o: myAlloc.s
	as $(ASFLAGS) -o myAlloc.o myAlloc.s
clean:
	rm -f *.o main
```

## Examples of results

As an example, a correct implementation of the main program should produce output similar to the following when executed with **"./main 8192 10"** (notice that the addresses may change):

```
Address                 Status          Size
0x76900b9a9000          free            1
0x76900b9a9009          free            2
0x76900b9a9013          free            4
0x76900b9a901f          free            8

Address                 Status          Size
0x76900b9a9000          busy            1
0x76900b9a9009          busy            2
0x76900b9a9013          free            4
0x76900b9a901f          busy            8
0x76900b9a902f          busy            10
```

## Delivery method

The file with the x86-64 assembly functions should be named *myAlloc.s*. This file should be submitted to Mooshak.